Processamento paralelo - Uma breve introdução

Vanderlei Júlio Debastiani (vanderleidebastiani@yahoo.com.br)

27 Maio 2020

Índice de conteúdo

1 Introdução

O pacote *parallel* é parte integrante do **R** e permite a execução de tarefas de maneira paralelizada usando múltiplos processos. Um exemplo típico de tarefas que podem ser realizadas de maneira paralela são simulações. Geralmente as simulações envolvem a geração de números aleatórios, então um conjunto de n simulações deve ser realizada usando os mesmos parâmetros. Ainda, conjuntos muito grandes de dados, que podem ser analisados separadamente, também podem ser analisados de maneira paralelizada. O ponto importante é que essas tarefas, ou blocos de códigos, não sejam relacionadas, ou seja, não precisam se comunicar de nenhuma maneira para serem concluídas e geralmente levam aproximadamente o mesmo período de tempo para serem finalizadas.

Computadores modernos, incluindo *laptops*, podem estar equipados como CPU (Unidade Central de Processamento - *Central Processing Unit*) capazes de realizar múltiplas tarefas. O número de processos que podem ser executados paralelamente depende do computador, mas mesmo computador com apenas um processador físico podem estar equipados com mais de um núcleo lógico. Teoricamente, cada um desses núcleos lógicos podem ser utilizados para realizar tarefas de maneira paralela.

O tutorial requer conhecimento sobre **R** e sobre função da família *Apply*. Os código são apenas brevemente comentados sendo que a ideia é que o leitor seja capaz de reproduzir os comandos aqui contidos, e busque mais informações sobre os argumentos extras nas páginas de ajuda de cada função. O texto apresneta apenas uma breve introdução ao processamento paralelo, ao final é apresentado um guia de ajuda rápida mostrando as principais funções apresentadas.

2 Terminologia e fluxo computacional

A terminologia do processamento paralelo difere entre aplicações (além de ser muito estranha). Um modelo básico é modelo mestre/trabalhadores que possui um processo **mestre** (*master*) que é chamado pelo usuário e é o responsável por iniciar os outros processos, denominados **trabalhadores** (*worker*). Esse conjunto de processos pode ser realizado em um ou vários computadores. Fisicamente os computadores que estão conectados e que trabalham em conjuntos são denominado de cluster (aglomerado) e os computadores que executam a mesma farefa são denominados de nó. Então, de maneira genérica, o conjunto de todos os processos de trabalho também pode ser chamado de **cluster** e cada processo desse pode ser chamado de **nó** (*node*).

O fluxo computacional básico de programação paralela é o seguinte:

- Iniciar m processos, fornecendo qualquer inicialização necessária. Cada processo será um trabalhador (*worker*);
- Enviar todos os dados necessários para a execução das tarefas aos trabalhadores;
- Dividir a tarefa em m pedaços, mais ou menos do mesmo tamanho, e enviar aos trabalhadores;
- Aguardar todos os trabalhadores concluírem suas tarefas e obter seus resultados;
- Encerrar os m processos.

3 Número de núcleos

A primeira coisa a se verificar para paralelizar qualquer tarefa é a disponibilidade de núcleos para realizar as tarefas. O número de núcleos pode ser determinado usando a função **detectCores**. Essa função tenta determinar o número de núcleos lógicos disponíveis, mas seu comportamento depende do sistema operacional (Windows, macOS, Linux) e versão desses sistemas.

4 Quando usar processamento paralelo

Em teoria, cada processador lógico adicional aumenta linearmente o rendimento do processamento, entretanto pode haver sobrecargas que reduzem a eficiência, de modo que os ganhos reais são geralmente inferiores aos teóricos. Ainda, é importante ter em mente que as etapas envolvidas na criação do cluster levam tempo: o cluster deve ser criado pelo sistema operacional, este deve ser configurado e as funções e dados precisam ser copiados para cada nó. De maneira geral, o processamento paralelo compensa quando o tempo gasto para configurar o cluster é bem inferior ao tempo ganho em processamento. Se o tempo gasto em processamento for curto e a etapas para configurar o cluster for longa, códigos executados de maneira paralela podem demorar mais tempo que os executados de maneira convencional.

5 Funções parApply

As funções do pacote *parallel* são análogas as funções da família *Apply*, basicamente usando os mesmos argumentos. A principal diferença é que as funções exigem também um objeto da classe *cluster* passado pelo argumento **cl**.

```
# Aplicar função nas margens de uma matriz
parApply(cl = NULL, X, MARGIN, FUN, ...)

# Aplicar função em cada linha (row) de uma matriz
parRapply(cl = NULL, x, FUN, ...)
```

```r
# Aplicar função em cada coluna (column) de uma matriz
parCapply(cl = NULL, x, FUN, ...)

# Aplicar função a cada elemento de um vetor
parSapply(cl = NULL, X, FUN, ..., simplify = TRUE)

# Aplicar função a cada elemento de um vetor, retornando lista
parLapply(cl = NULL, X, fun, ...)
```

6 Clusters e Forking

No pacote *palallel* há dois tipos básicos de paralização, **"PSOCK"** e **"FORK"**, sendo que:

- **PSOCK** - Disponível em todas as plataformas R e criado pela função **makePSOCKcluster**. É similar à iniciar uma cópia adicional do R, os processos executados até o momento não são passados para as cópias, ou seja, para os nós.
- **FORK** - Via bifurcação (*Fork*) disponível em todas as plataformas R, exceto no Windows, e criado pela função **makeForkCluster**. Cria um novo processo R fazendo uma cópia completa do processos executados até então pelo mestre, incluindo a área de trabalho (*workspace*) e estado do fluxo de números aleatórios.

A função **makeCluster** cria os cluster de ambos tipos de paralização, basta especificar o tipo (**"PSOCK"** ou **"FORK"**) no argumento **type**. Após todas as tarefas serem concluídas o cluster deve ser encerrado, ou seja os processos dos trabalhadores encerrados, pela função **stopCluster**.

Após a criação do cluster, é preciso enviar todas as funções e dados necessários para a execução das tarefas pelos trabalhadores. A função **clusterExport** atribui objetos e função locais do processo mestre para os nós do cluster, ou seja, a função torna disponível para cada trabalhador objetos e funções disponíveis até então apenas na área de trabalho (*workspace*). A função **clusterEvalQ** atribui uma expressão literal em cada nó do cluster, sendo usada para carregar pacotes ou expressões nos nós do cluster. Entretando, as funções dos pacotes são chamadas diretanemente dos *namespaces* de cada pacote quando a atribuição é feita usando **::**, isso permite que os pacotes não precisem ser carregador em cada nó do cluster usando a função **clusterEvalQ**.

```r
# Gerar o cluster
makeCluster(spec, type, ...)

# Exportar objetos e funções para os nós do cluster
clusterExport(cl = NULL, varlist, envir = .GlobalEnv)

# Atribuir uma expressão em cada nós do cluster
clusterEvalQ(cl = NULL, expr)

# Encerrar o cluster
stopCluster(cl = NULL)
```

```r
require(parallel)

# Dados de exemplos
VETOR.TESTE.PAR <- c(1, 1, 1, 1)
VETOR.TESTE.PAR
[1] 1 1 1 1
LISTA.TESTE.PAR <- lapply(1:4, function(x) matrix(runif(16), nrow = 4, ncol = 4))
```

```
LISTA.TESTE.PAR
[[1]]
           [,1]       [,2]      [,3]       [,4]
[1,] 0.06817543 0.56159274 0.6864651 0.92121096
[2,] 0.77045045 0.61352461 0.5341523 0.68350637
[3,] 0.99953591 0.60233965 0.3078345 0.82898813
[4,] 0.44568205 0.01838455 0.6325704 0.05268821

[[2]]
           [,1]      [,2]      [,3]       [,4]
[1,] 0.56458270 0.1450991 0.6516914 0.71261855
[2,] 0.40819983 0.1028881 0.7962492 0.02283342
[3,] 0.02259288 0.3180062 0.3063864 0.19997245
[4,] 0.35562133 0.8666211 0.1888149 0.13162242

[[3]]
            [,1]      [,2]      [,3]      [,4]
[1,] 0.727792480 0.4158553 0.8991458 0.5034986
[2,] 0.510698451 0.3896529 0.7620215 0.5303271
[3,] 0.003885456 0.4558392 0.3224969 0.9330110
[4,] 0.258794836 0.2592317 0.3108774 0.4417620

[[4]]
          [,1]      [,2]      [,3]      [,4]
[1,] 0.1640229 0.6943789 0.8977924 0.5339227
[2,] 0.0641928 0.6100107 0.9620942 0.5423346
[3,] 0.7322833 0.7772360 0.6394542 0.1367078
[4,] 0.9080437 0.5956990 0.1905694 0.7015909


# Detectar número de núcleos lógicos
detectCores()
[1] 4


##  Exemplo 1
# Funcionamento básico

# Criar cluster com 2 nós do tipo "PSOCK"
cl <- makeCluster(2, type = "PSOCK")
cl
socket cluster with 2 nodes on host 'localhost'

# Aplicar ao vetor alguma função de maneira paralela ...
parSapply(cl, X = VETOR.TESTE.PAR, FUN = length)
[1] 1 1 1 1

# ... outro exemplo
parSapply(cl, X = LISTA.TESTE.PAR, FUN = sum)
[1] 8.727101 5.793800 7.724890 9.150333

# Encerrar o cluster
stopCluster(cl)
```

```
## Exemplo 2
# Quando há necessidade de uma função de um pacote que precisa ser carregado

# Função que utiliza a função rational do pacote MASS
require(MASS)
f.par <- function(X, ...){
    res <- rational(solve(X, X), ...)
    return(res)
}

# Criar cluster com 2 nós do tipo "PSOCK"
cl <- makeCluster(2, type = "PSOCK")
cl
socket cluster with 2 nodes on host 'localhost'

# O pacote MASS não está disponível nos nós ...
parSapply(cl, X = LISTA.TESTE.PAR, FUN = f.par, simplify = FALSE)
Error in checkForRemoteErrors(val): 2 nodes produced errors; first error: could not find function "rati

# ... é preciso carregar o pacote em cada nó, enviando por meio de uma expressão ...
clusterEvalQ(cl, expr = require(MASS))
[[1]]
[1] TRUE

[[2]]
[1] TRUE

# ... então pacote ficará disponível
parSapply(cl, X = LISTA.TESTE.PAR, FUN = f.par, simplify = FALSE)
[[1]]
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1

[[2]]
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1

[[3]]
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1

[[4]]
     [,1] [,2] [,3] [,4]
```

```
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1

# Encerrar o cluster
stopCluster(cl)


## Exemplo 3
# Quando há necessidade de uma função de um pacote que precisa ser carregado usando ::

# Função que utiliza a função rational do pacote MASS atribuída com ::
f.par <- function(X, ...){
    res <- MASS::rational(solve(X, X), ...)
    return(res)
}

# Criar cluster com 2 nós do tipo "PSOCK"
cl <- makeCluster(2, type = "PSOCK")
cl
socket cluster with 2 nodes on host 'localhost'

# A função fica disponível sem a nescessidade da função clusterEvalQ
parSapply(cl, X = LISTA.TESTE.PAR, FUN = f.par, simplify = FALSE)
[[1]]
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1

[[2]]
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1

[[3]]
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1

[[4]]
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
```

```r
# Encerrar o cluster
stopCluster(cl)


## Exemplo 4
# Quando a função aplicada requer outra função da área de trabalho (workspace)

# Função da área de trabalho
f.par.interna <- function(X){
    nr <- nrow(X)
    res <- X/nr
    return(res)
}

# Função que será aplicada
f.par2 <- function(X, ...){
    X2 <- f.par.interna(X)
    res <- MASS::rational(solve(X, X2), ...)
    return(res)
}

# Criar cluster com 2 nós do tipo "PSOCK"
cl <- makeCluster(2, type = "PSOCK")

# A função f.par.interna não está disponível nos nós ...
parSapply(cl, X = LISTA.TESTE.PAR, FUN = f.par2, simplify = FALSE)
Error in checkForRemoteErrors(val): 2 nodes produced errors; first error: could not find function "f.pa

# ... então é preciso exportar a função ...
clusterExport(cl, varlist = "f.par.interna")

# ... tornondo-a disponível
parSapply(cl, X = LISTA.TESTE.PAR, FUN = f.par2, simplify = FALSE)
[[1]]
     [,1] [,2] [,3] [,4]
[1,] 0.25 0.00 0.00 0.00
[2,] 0.00 0.25 0.00 0.00
[3,] 0.00 0.00 0.25 0.00
[4,] 0.00 0.00 0.00 0.25

[[2]]
     [,1] [,2] [,3] [,4]
[1,] 0.25 0.00 0.00 0.00
[2,] 0.00 0.25 0.00 0.00
[3,] 0.00 0.00 0.25 0.00
[4,] 0.00 0.00 0.00 0.25

[[3]]
     [,1] [,2] [,3] [,4]
[1,] 0.25 0.00 0.00 0.00
[2,] 0.00 0.25 0.00 0.00
[3,] 0.00 0.00 0.25 0.00
[4,] 0.00 0.00 0.00 0.25
```

```
[[4]]
     [,1] [,2] [,3] [,4]
[1,] 0.25 0.00 0.00 0.00
[2,] 0.00 0.25 0.00 0.00
[3,] 0.00 0.00 0.25 0.00
[4,] 0.00 0.00 0.00 0.25

# Encerrar o cluster
stopCluster(cl)
```

7 Gerador de número aleatórios

Quando as funções envolvem geração de número aleatórios é preciso tomar cuidados adicionais. Os tipos básicos de cluster para paralização **"PSOCK"** e **"FORK"** diferem em comportamento da geracão de números aleatórios. Cluster do tipo **"PSOCK"** inicializam com seed do gerador de números aleatórios independentes, já os do tipo **"FORK"** usam o mesmo seed para todos os nós. A função **clusterSetRNGStream** pode ser usada para atribuir um inicializador do gerador de número aleatórios em todos os nós do cluster.

```
## Exemplo 5
# Cluster do tipo "PSOCK" e gerador de números aleatórios

# Criar cluster com 2 nós do tipo "PSOCK"
cl <- makeCluster(2, type = "PSOCK")

# Note que cada nó do cluster inicializa com um seed, sem qualquer controle sobre o seed
parSapply(cl, X = VETOR.TESTE.PAR, FUN = function(x) rnorm(x))
[1] -1.34109311  1.94068545 -0.06381917  0.54483779
parSapply(cl, X = VETOR.TESTE.PAR, FUN = function(x) rnorm(x))
[1] -1.1436287  1.9521995 -0.7966862  1.6565908

# Encerrar o cluster
stopCluster(cl)

## Exemplo 6
# Cluster do tipo "PSOCK" e gerador de números aleatórios

# Criar cluster com 2 nós do tipo "PSOCK"
cl <- makeCluster(2, type = "PSOCK")

# Atribuit um seed para cada nó do cluster
# Note que ao resetar o seed os valores sorteador são os mesmos
clusterSetRNGStream(cl, 123)
parSapply(cl, X = VETOR.TESTE.PAR, FUN = function(x) rnorm(x))
[1] -0.9685927  0.7061091 -0.4094454  0.8909694
clusterSetRNGStream(cl, 123)
parSapply(cl, X = VETOR.TESTE.PAR, FUN = function(x) rnorm(x))
[1] -0.9685927  0.7061091 -0.4094454  0.8909694

# Encerrar o cluster
stopCluster(cl)

## Exemplo 7
```

```r
# Cluster do tipo "FORK" e gerador de números aleatórios

# Criar cluster com 2 nós do tipo "FORK"
cl <- makeCluster(2, type = "FORK")

# Note que cada nó parte com o seed inicializado na área de trabalho
# O primeiro e o terceiro números são iguas, assim como o segundo e o quarto...
parSapply(cl, X = VETOR.TESTE.PAR, FUN = function(x) rnorm(x))
[1] -0.3939316 -0.2834549 -0.3939316 -0.2834549
parSapply(cl, X = VETOR.TESTE.PAR, FUN = function(x) rnorm(x))
[1]  0.4240360 -0.1215351  0.4240360 -0.1215351
# Note ainda que a sequência de valores sorteadas é a mesma fora do cluster
rnorm(VETOR.TESTE.PAR)
[1] -0.3939316 -0.2834549  0.4240360 -0.1215351

# Encerrar o cluster
stopCluster(cl)

## Exemplo 8
# Cluster do tipo "FORK" e gerador de números aleatórios

# Criar cluster com 2 nós do tipo "FORK"
cl <- makeCluster(2, type = "FORK")

# Atribuir um seed para cada nó do cluster, diferente da área de trabalho
# Note que ao resetar o seed os valores sorteador são os mesmos
clusterSetRNGStream(cl, 123)
parSapply(cl, X = VETOR.TESTE.PAR, FUN = function(x) rnorm(x))
[1] -0.9685927  0.7061091 -0.4094454  0.8909694
clusterSetRNGStream(cl, 123)
parSapply(cl, X = VETOR.TESTE.PAR, FUN = function(x) rnorm(x))
[1] -0.9685927  0.7061091 -0.4094454  0.8909694
# Note que o seed da área de trabalho permance diferente nos aplicados nos nós
rnorm(VETOR.TESTE.PAR)
[1] -1.3967680  2.2733335  1.0089337 -0.6220876

# Encerrar o cluster
stopCluster(cl)
```

8 Guia de ajuda rápida

```
# Parallel
detectCores() - Contar número de núcleos lógicos
parApply() - Aplicar função nas margens de uma matriz
parRapply() - Aplicar função em cada linha de uma matriz
parCapply() - Aplicar função em cada coluna de uma matriz
parSapply() - Aplicar função a cada elemento de um vetor
parLapply() - Aplicar função a cada elemento de um vetor, retornando lista
makeCluster() - Gerar cluster
clusterExport() - Exportar objetos e funções para os nós do cluster
clusterEvalQ() - Avaliar uma expressão os nós do cluster
clusterSetRNGStream() - Controlar gerador de número aleatórios do cluster
stopCluster() - Encerrar cluster
```

9 Conclusão

O objetivo deste texto foi apenas apresentar uma breve introdução ao processamento paralelo no R com o pacote **parallel**, muitas outras funções e opções estão disponívels no pacote. Espero que este texto tenha sido útil e, por favor, avise-me se tiver dúvidas ou sugestões sobre este texto.

10 Mais informações

Outros textos e tutoriais sobre R podem ser encontrados em https://vanderleidebastiani.github.io/tutoriais.

11 Referências

R Core Team. 2020. **R: A language and environment for statistical computing.** https://www.R-project.org/.

R Core Team. 2020. **Package 'parallel'.** https://www.R-project.org/.